HPC

High Performance Computing (HPC) refers to the use of **supercomputers and parallel processing techniques** to solve complex computational problems quickly and efficiently.

🔹 What is OpenMP?

OpenMP (Open Multi-Processing) is an **API (Application Programming Interface)** that supports **multi-platform shared memory multiprocessing programming** in C, C++, and Fortran. It allows developers to write parallel code that can utilize **multiple cores of a CPU** effectively using a simple set of compiler directives (like #pragma in C/C++) and library routines.

OpenMP is used to:

- **Speed up execution** of programs by performing tasks **in parallel**.
- Make better use of **multi-core CPUs**.
- Reduce execution time of compute-intensive tasks (e.g., matrix multiplication, numerical simulations).
- Simplify parallel programming with a high-level interface (no need for low-level thread handling).

⇨ **g++ -fopenmp your_code.cpp -o your_program**

> g++ -fopenmp name.cpp -o search
> ./search

- -fopenmp: This flag **enables OpenMP support** in the compiler.
- -o your_program: Specifies the **output filename** for the compiled program.

🔹 Why this is used:

This command tells the compiler to compile your C++ code with OpenMP support, allowing it to recognize and handle OpenMP directives (like #pragma omp parallel).

🔹 #pragma omp parallel

#pragma omp parallel is a **directive** in OpenMP that tells the compiler to **create multiple threads** to execute the following block of code **in parallel**.

preprocessor directives:

- #include <iostream>:

This line includes the iostream library, which provides standard input/output functionalities. It allows the program to interact with the user through the console, using objects like cin for input and cout for output.

- #include <omp.h>:

This line includes the OpenMP library, which provides support for parallel programming. It enables the program to utilize multiple threads for concurrent execution, potentially speeding up computationally intensive tasks.

- using namespace std;

This line brings the standard namespace into the current scope. The standard namespace contains commonly used C++ components like cin, cout, and various data types. By using this line, you can use these components directly without explicitly specifying the std:: prefix.

1. BFS DFS:

Breadth-First Search

Breadth-First Search (or BFS) is an algorithm for searching a tree or an undirected graph data structure. Here, we start with a node and then visit all the adjacent nodes in the same level and then move to the adjacent successor node in the next level. This is also known as level-by-level search.

Queue: FIFO (First In, First Out)

Enqueue: Add an element at the rear.

Dequeue: Remove an element from the front.

Depth-First Search:

DFS is an algorithm for searching a tree or an undirected graph data structure. Here, the concept is to start from the starting node known as the root and traverse as far as possible in the same branch.

Stack: LIFO (Last In, First Out)

Push: Add an element at the top.

Pop: Remove an element from the top.

2. Bubble Merge sort:

3. MIN MAX SUM AVG:

Parallel reduction :

refers to algorithms which combine an array of elements producing a single value as a result. by leveraging multiple processors or threads simultaneously. Problems eligible for this algorithm include those which involve operators that are associative and commutative in nature. Some of them include. Sum of an array. Minimum/Maximum of an array.

- #include <climits>:

This line includes the climits library, which defines various limits for integer data types. It provides constants like INT_MAX and INT_MIN representing the maximum and minimum values that an integer variable can hold.

✅ 1. Basic Viva Questions (Understanding Concepts)

Question	What to Answer
❓ What is OpenMP?	OpenMP is an API for writing multi-threaded parallel programs in C, C++, and Fortran. It uses compiler directives like #pragma omp.
❓ What is parallel BFS/DFS?	BFS and DFS can be executed in parallel by processing multiple nodes at the same depth or stack level using threads.
❓ Why did you use a tree?	Trees are structured and simpler for demonstration. It avoids cycles and makes parallel traversal easier.
❓ How did you parallelize BFS/DFS?	I used #pragma omp parallel for to parallelize the loop that processes nodes/stack/queue items. Critical sections protect shared resources like the queue or stack.
❓ What is a critical section in OpenMP?	It prevents data races by allowing only one thread at a time to access the enclosed code block.
❓ How many threads are used?	It depends on system threads. By default, OpenMP uses all available threads. I can also set them manually using omp_set_num_threads(n);.

✅ 2. Questions Specific to Your Code

Question	Your Answer (based on the code)
❓ What data structure did you use to represent the tree?	I used a Node class with left, right, and data.
❓ How is the tree built?	The user inputs node values level-by-level using a queue (createTree() method).
❓ What does bfs() do?	It performs breadth-first traversal using a queue and OpenMP parallelization for visiting nodes at each level.
❓ What does dfs() do?	It uses a stack to perform depth-first traversal with OpenMP parallel sections.
❓ Why did you use #pragma omp critical?	To prevent race conditions when accessing or modifying the shared queue/stack.
❓ What happens if you remove the critical section?	Multiple threads might access and modify the queue/stack simultaneously, leading to incorrect results or crashes.
❓ Why is time calculated?	To compare performance and demonstrate speed of traversal under parallel execution.

✅ 3. Output-Based Questions

Question	Tip
❓ What is the output of BFS for this tree?	Be ready to trace BFS step-by-step (use example from earlier).
❓ What is the output of DFS?	Similarly, explain how stack is used to visit left and right children.
❓ What is the advantage of BFS over DFS in trees?	BFS is better for shortest path; DFS is better for memory if tree is deep.

✅ 4. Process Flow — How to Explain

Here's a clean way to describe your code in viva:

"My program constructs a binary tree using user input level-by-level. Then it performs BFS and DFS traversals. I use a queue for BFS and a stack for DFS. I apply OpenMP to parallelize the traversal loops. #pragma omp parallel for allows multiple threads to process nodes at the same level (BFS) or depth (DFS). I use #pragma omp critical to protect shared structures like queue or stack from data races. I also measure the time taken for each traversal using omp_get_wtime()."

✅ Bonus: Questions if Viva Goes Deeper

Question	Answer
❓ Can BFS/DFS be fully parallel?	Not easily — because order matters and tree shape limits concurrency. But level-by-level or node processing can be partially parallel.
❓ What are data races?	When multiple threads access shared data at the same time and at least one modifies it, leading to unpredictable behavior.
❓ What happens if you use #pragma omp parallel instead of parallel for?	parallel creates threads, but parallel for divides loop iterations automatically among them.

✅ **What You Must Know About the Code**

Make sure you clearly understand:

- Node structure
- createTree() logic (level-order)
- Why queue in BFS and stack in DFS
- Why and where #pragma omp parallel for and #pragma omp critical are used
- How traversal order works
- Output for a given tree input
- Compilation: g++ -fopenmp yourcode.cpp -o yourcode